REPORT DOCUMENTATION PAGE

Form Approved
OMB NO. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Reprint

4. TITLE AND SUBTITLE

TITLE ON REPRINT

5. FUNDING NUMBERS

ARO_MIPR

6. AUTHOR(S)

AUTHOR(S) ON REPRINT

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(ES)

Naval Postgraduate School

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

ARO 40473.30-MA-SP

11. SUPPLEMENTARY NOTES

The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision, unless so designated by other documentation.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12 b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

ABSTRACT ON REPRINT

14. SUBJECT TERMS	15. NUMBER IF PAGES
	16. PRICE CODE

17. SECURITY CLASSIFICATION OR REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Lecture Notes in Computer Science 1966

Subhash Bhalla (Ed.)

Databases in Networked Information Systems

International Workshop DNIS 2000
Aizu, Japan, December 2000
Proceedings

Springer

JAVA Wrappers for Automated Interoperability

Ngom Cheng, Valdis Berzins, Luqi, and Swapan Bhattacharya

Department of Computer Science
Naval Postgraduate School
Monterey, CA. 93943 USA
{cheng, berzins, luqi, swapan}@cs.nps.navy.mil

Abstract. This paper concentrates on the issues related to implementation of interoperability between distributed subsystems, particularly in the context of re-engineering and integration of several centralized legacy systems. Currently, most interoperability techniques require the data or services to be tightly coupled to a particular server. Furthermore, as most programmers are trained in designing stand-alone application, developing distributed system proves to be time-consuming and difficult. Here, we addressed those concerns by creating an interface wrapper model that allows developers to treat distributed objects as local objects. A tool that automatically generates the features of Java interface wrapper from a specification language called the Prototyping System Description Language has been developed based on the model.

1 Introduction

Interoperability between software systems is the ability to exchange services from one system to another. In order to exchange services, commands and data are relayed from the requesters to the service providers. Current business and military systems are typically 2-tier or 3-tier systems involving clients and servers, each running on different machines in the same or different locations. Current approaches for n-tier systems have no standardization of protocol, data representation, invocation techniques etc. Other problems related to interoperability are the implementation of distributed systems and the use of services from heterogeneous operating environments. These include issues concerning sharing of information amongst various operating systems, and the necessity for evolution of standards for using data of various types, sizes and byte ordering, in order to make them suitable for interoperation. These problems make interoperable applications difficult to construct and manage.

1.1 Current State-of-the-Art Solutions

Presently, the solutions attempting to address these interoperability problems range from low-level sockets and messaging techniques to more sophisticated middleware technology like object resource brokers (CORBA, DCOM). Middleware technology uses higher abstraction than messaging, and can simplify the construction of interoperable applications. It provides a bridge between the service provider and

S. Bhalla (Ed.): DNIS 2000, LNCS 1966, pp. 45–64, 2000.

requester by providing standardized mechanisms that handle communication, data exchange and type marshalling. The implementation details of the middleware are generally not important to developers building the systems. Instead, developers are concerned with service interface details. This form of information hiding enhances system maintainability by encapsulating the communication mechanisms and providing stable interface services for the developers. However, developers still need to perform significant work to incorporate the middleware's services into their systems. Furthermore, they must have a good knowledge of how to deploy the middleware services to fully exploit the features provided.

Current middleware approaches have another major limitation in design - the data and services are tightly coupled to the servers. Any attempt to parallelize or distribute a computation across several machines therefore encounters complicated issues due to this tight control of the server process on the data. Tuning performance by redistributing processes and data over different hardware configurations requires much more effort for software adjustment than system administrators would like.

1.2 Motivation

Distributed data structures provide an entirely different paradigm. Here, data is no longer coupled to any particular process. Methods and services that work on the data are also uncoupled from any particular process. Processes can now work on different pieces of data at the same time. Until recently, building distributed data structures together with their requisite interfaces has proved to be more daunting than other conventional interoperability middleware techniques. The arrival of JavaSpace has changed the scenario to some extent. It allows easy creation and access to distributed objects. However, issues concerning data getting lost in the network, duplicated data items, out-dated data, external exception handling and handshaking communication between the data owner and data users are still open. Developers have to devise ways to solve those problems and standardize them between applications.

1.3 Proposal

The situation concerning interoperability would greatly improve if a developer working on some particular application could treat distributed objects as local objects within the application. The developers could then modify the distributed object as if it is local within the process. The changes may, however, still need to be reflected in other applications using that distributed object without creating any problems related to inconsistency. The current research aims at attaining this objective by creating a model of an interface wrapper that can be used for a variety of distributed objects. In addition, we seek models that can automate the process of generating the interface wrapper directly from the interface specification of the requirement, thereby greatly improving developers' productivity.

A tool, named the Automated Interface Codes Generator (AICG), has been developed to generate the interface wrapper codes for interoperability, from a specification language called the Prototype System Description Language (PSDL) [9]. The tool

uses the principles of distributed data structure and JavaSpace Technology to encapsulate transaction control, synchronization, and notification together with lifetime control to provide an environment that treats distributed objects as if there were local within the concerned applications.

2 Review of Previous Works

A basic idea for enhancing interoperability is to make the network transparent to the application developers. Previous approaches [1] include 1) Building blocks for interoperability, 2) Architectures for unified, systematic interoperability and 3) Packaging for encapsulating interoperability services. These approaches have been assessed and summerized using Kiviat graphs by Berzins [1] with various weight factors. The Kiviat graphs give a good summary of the strong and weak points of various approaches. ORBs and Jini are currently among the promising technologies for interoperability.

2.1 ORB Approaches

There are however, some concerns with the ORB models. Sullivan [13] provides a more in-depth analysis of the DCOM model, highlighting the architecture conflicts between Dynamic Interface Negotiation (how a process queries a COM interface and its services) and Aggregation (component composition mechanism). Interface negotiation does not function properly within the aggregated boundaries. This problem arises because interacting components share an interface. An interface is shared if the constructor or QueryInterface functions of several components can return a pointer to it. QueryInterface rules state that a holder of a shared interface should be able to obtain interfaces of all types appearing on both the inner and outer components. However, an aggregator can refuse to provide interfaces of some types appearing on an inner component by hiding the inner component. Thus, QueryInterface can fail to work properly with respect to delegation to the inner interface.

Hence, for the ORB approaches, detailed understanding of the techniques is required to design a truly reliable interoperable system. Programmers however, are trained mostly on standalone programming techniques. Adding specialized network programming models increases the learning as well as development time, with occasional slippage of target deadlines. Furthermore, bugs in distributed programs are harder to detect and consequences of failure are more catastrophic. An abnormal program can cause other programs to go astray in a connected distributed environment [9], [12].

2.2 Prototyping

The demand for large, high quality systems has increased to the point where a quantum change in software technology is needed [9]. Requirements and

specification errors are a major cause of faults in complex systems. Rapid prototyping is one of the most promising solutions to this problem. Completely automated generation of prototype from a very high-level language is feasible and generation of skeleton programming structures is currently common in the computer world. One major advantage of the automatic generation of codes is that it frees the developers from the implementation details by executing specifications via reusable components [9].

In this perspective, an integrated software development environment, named Computer Aided Prototyping System (CAPS) has been developed at the Naval Postgraduate School, for rapid prototyping of hard real-time embedded software systems, such as missile guidance systems, space shuttle avionics systems, software controllers for a variety of consumer appliances and military Command, Control, Communication and Intelligence (C3I) systems [11]. Rapid prototyping uses rapidly constructed prototypes to help both the developers and their customers visualize the proposed system and assess its properties in an iterative process. The heart of CAPS is the Prototyping System Description Language (PSDL). It serves as an executable prototyping language at a specification and software architecture level and has special features for real-time system design. Building on the success of computer aided rapid prototyping system (CAPS) [11], the AICG model also uses PSDL for specification of distributed systems and automates the generation of interface codes with the objective of making the network transparent from the developer's point of view.

2.3 Transaction Handling

Building a networked application is entirely different from building a stand-alone system in the sense that many additional issues need to be addressed for smooth functioning of a networked application. The networked systems are also susceptible to partial failures of computation, which can leave the system in an inconsistent state.

Proper transaction handling is essential to control and maintain concurrency and consistency within the system. Yang has examined the limitation of hard-wiring concurrency control into either the client or the server. He found that the scalability and flexibility of these configurations is greatly limited. Hence, he presented a middleware approach: an external transaction server, which carries out the concurrency control policies in the process of obtaining the data. Advantages of this approach are 1) transaction server can be easily tailored to apply the desired concurrency control policies of specific client applications. 2) The approach does not require any changes to the servers or clients in order to support the standard transaction model. 3) Coordination among the clients that share data but have different concurrency control policies is possible if all of the clients use the same transaction server.

The AICG model uses the same approach, by using an external transaction manager such as the one provided by SUN in the JINI model. All transactions used by the clients and servers are created and overseen by the manager.

3 The AICG Interaction Model

The AICG model encapsulates some of the features of JavaSpace and Jini to provide a simplified ways of developing distributed applications.

3.1 Jini Model

The Jini model is designed to make a service on a network available to anyone who can reach it, and to do so in a type-safe and robust way [4]. The ability of Jini model is based on five key concepts: (1) *Discovery* is the process used to find communities on the network and join with them. (2) *Lookup* governs how the code that is needed to use a particular services finds its way into participants that want to use that service. (3)*Leasing* is the technique that provides the Jini self recovering ability. (4) *Remote events* allow services to notify each other of changes to their state (5) *Transactions* ensure that computations of several services and their host always remain in "safe" state.

The Jini model was designed by Sun Microsystems with simplicity, reliability and scalability as the focus. Its vision is that Jini-enable devices such as PDA, cell phone or a printer, when plugged into a TCP/IP network, should be able to automatically detect and collaborate with other Jini-enabled devices.

The powerful features of Jini provide a good groundwork for developing interoperability systems. However, the lack of automation for creating interface software and the need for developers to fully understand the Jini Model before they can use it created the same problems for developers as other interoperability approaches.

3.2 The JavaSpace Model

The JavaSpace model is a high-level coordination tool for gluing processes together in a distributed environment. It departs from conventional distribution techniques using message passing between processes or invoking methods on remote objects. The technology provides a fundamentally different programming model that views an application as a collection of processes cooperating via the flow of freshly copied objects into and out of one or more spaces. This space-based model of distributed computing or distributed structure has its roots in the Linda coordination language [3] developed by Dr. David Gelernter at Yale University.

3.2.1 Distributed Data Structure and Loosely Coupled Programming

Conceptually a distributed data structure is one that can be accessed and manipulated by multiple processes at the same time without regard for which machine is executing those processes. In most distributed computing models, distributed data structures are hard to achieve. Message passing and remote method invocation systems provide a good example of the difficulty. Most of the systems tend to keep data structure behind one central manager process, and processes that want to perform work on the data

structure must "wait in line" to ask the manager process to access or alter a piece of data on their behalf. Attempts to parallelize or distribute a computation across more than one machine face bottlenecks since data are tightly coupled by the one manager process. True concurrent access is rarely achievable.

Distributed data structures provide an entirely different approach where we uncouple the data from any particular process. Instead of hiding data structure behind a manager process, we represent data structures as collections of objects that can be independently and concurrently accessed and altered by remote processes. Distributed data structures allow processes to work on the data without having to wait in line if there are no serialization issues.

3.2.2 Space

A space is a shared, network-accessible repository for objects. Processes use the repository as a persistent object storage and exchange mechanism Processes perform simple operations to write new objects into space, take objects from space, or read (make a copy of) objects in a space. When taking or reading objects, processes use a simple value-matching lookup to find the objects that matter to them. If a matching object is not found immediately, then a process can wait until one arrives. Unlike conventional object stores, processes do not modify objects in the space or invoke their methods directly. To modify an object, a process must explicitly remove it, update it, and reinsert it into the space. During the period of updating, other processes requesting for the object will wait until the process writes the object back to the space. This protocol for modification ensures synchronization, as there can be no way for more than one process to modify an object at the same time. However, it is possible for many processes to read the same object at the same time.

Key Features of JavaSpace:

- Spaces are persistent: Spaces provide reliable storage for objects. Once stored in the space, an object will remain there until a process explicitly removes it.
- Spaces are transactionally secure: The Space technology provides a transaction model that ensures that an operation on a space is atomic. Transactions are supported for single operations on a single space, as well as multiple operations over one or more spaces.
- Spaces allow exchange of executable content: While in the space, objects are just passive data, however, when we read or take an object from a space, a local copy of the object is created. Like any other local object; we can modify its public fields as well as invoke its methods.

3.3 The AICG Approach

The AICG approach to interoperability has two parts. The first part is to develop a model to completely hide the interoperability from the developers and the second part of the approach is to design a tool that automates the process of integrating the AICG model into the distributed application so as to aid the development process.

3.3.1 The AICG Model

The AICG model is built on JavaSpace and Jini. It is designed to wrap around data structures or objects that are shared between concurrent applications across a network. The model gives the applications complete access to the contents of the objects as though they were the sole owners of the data. Synchronization, transaction and error handling are built into the model, freeing the developers to concentrate on the actual requirement of the applications.

AICG uses the JavaSpace Distributed Data Structure principles as the main communication channel for exchange of services. The model also encompasses Jini services like Transaction, Leasing and Remote Event. However, the difference is that the model wraps the services provided by the JavaSpace and Jini and hide their usage from the application. Developers are not required to understand the underlying principles before they can use the model. They should however be aware of object oriented programming constraints such as no direct access to the attributes of an object is allowed without going through the object methods.

The most common use of the AICG model is to encapsulate objects that are to be shared. This form of abstraction has an advantage over direct use of a JavaSpace. The JavaSpace distributed protocol for modification ensures synchronization by enforcing that a process wishing to modify the object has to physically remove it from the space, alter it and write it back to the space. There can be no way for more than one process to modify an object at the same time. However, this does not prevent other processes from overwriting the updated data. For example, in an ordinary JavaSpace, the programmer of Process A could specify a "read" operation, followed by a "write" operation. This would result in 2 copies of the object in the Space. The AICG model prevents this since the 3 basic commands are embedded into distributed objects that are automatically generated to conform to the proper protocol. All modifications on the object are automatically translated to "take", followed by "write" and all operations that access the fields of the distributed object are translated to "read". These ensure that local data are up-to-date and serialization is maintained.

Although the basic idea of the AICG model is simple, it requires many supporting features to make it work. Distributed objects may be lost if a process removes them from the space and subsequently crashes or is cut off from the network. Similarly, the system may enter a deadlock state if processes request more than one distributed object while, at the same time, holding on to distributed objects required by other processes. Similarly, latency and performance are very different between local access and remote access. Those issues should not be ignored in any interoperability techniques, if the systems to be built using the techniques must be robust. ORB techniques such as RPC and CORBA do not even consider performance and latency as part of their programming model, they treat it as a "hidden" implementation detail that programmer must implicitly be aware of and deal with while they preach that accessing remote object is similar to accessing local object.

The AICG model has a set of four supporting modules to handle those situations. These modules provide transaction handling and user-defined latency to ensure integrity of the updates, exception handling for reporting errors and failures without

crashing the system, a notification channel to inform the application of certain events, and lease control for freeing up unused object during "house keeping". The supporting features are discussed in section 5.

3.3.2 The AICG Tool

The second part of the research aims at developing a tool that generates software wrapper realizing the AICG model to aid the construction of distributed applications. The tool is designed to generate interface wrappers for data structures or objects that need to be shared, and is particularly useful for applications that can be modeled as flows of objects through one or more servers. The tool allows the developers to use all the features in the AICG model without the need to write complicated codes. This enhances interoperability by making network and concurrent issues transparent to the application developers.

The interface wrappers are generated from an extension of a prototype description language called Prototyping System Description Language (PSDL). The extended Description language (PSDL-ext) expands property definitions that are specific only to AICG model.

Some of the salient features of the AICG model generated by the tool are:

- Distributed objects are treated as local objects within the application process. The application code need not depend on how the object is distributed, since the local object copy is always synchronous with the distributed copy.
- Synchronization with various applications is automatically handled. Since the AICG model is based on the space transaction secure model and all operations are atomic. Deadlock is prevented automatically within the interface and each object has through transaction control. Any type of object can be shared as long as the object is serializable. Any data structure and object can be distributed as long as it obeys and implements the java serializable feature.
- Every distributed object has a lifetime. The distributed object lifetime is a period of time guaranteed by the AICG model for storage and distribution of the object. The time can be set by developer.
- All write operations are transaction secure by default. AICG transactions are based on the Atomicity, Consistency, Isolation, and Durability (ACID) features.
- Clients can be informed of changes to the distributed object through the AICG event model. A client application can subscribe for change notification, and when the distributed object is modified, a separate thread is spawned to execute a callback method defined by the developer.
- The wrapper codes are generated from high-level descriptive languages; hence, they are more manageable and more maintainable.

4 Types of Services

Services can be basic raw data, messages, remote method invocation, complex data structures, or object with attributes and methods. The AICG model is suited for exchange and sharing of complex data structures and objects. It can be tailored for raw data, messaging, and remote method invocation types of communication.

The AICG model uses the space as a transmission medium and hence loosens the tie between producers and consumers of services which are forced to interact indirectly through a space. This is a significant difference, as loosely coupled systems tend to be more flexible and robust.

4.1 Overview of the PSDL Interface

Prototype System Description Language (PSDL) provides a data flow notation augmented by application-orientated timing and control constraints to describe a system as a hierarchy of networks of processing units communicating via data streams [1]. Data Streams carry values of abstract types and provide error-free communication channels. PSDL can be presented in a semi-graphical form for easy specifying of the specifications and requirements. An introduction to the real-time aspects of the PSDL can be found in [1] and [2].

In PSDL, every computational entity such as an application, a procedure, a method or a distributed system is represented as an operator. It is hierarchical in nature and each operator can be decomposed to sub-operators and streams. Every operator is a state machine. Its internal states are modeled by variable sets local only to this operator. Operators are represented as circular icons in PSDL Graph, and triggered by data stream or periodic timing constraints. When an operator is triggered, it reads one data value from each input stream and computes the results if the execution guard or constraint is satisfied. The results are placed on the output streams if the output guard is satisfied.

Operators communicate via data streams. These data streams contain values that are instances of an abstract data type. For each stream, there are zero or more operators that write data on the stream and zero or more operators that read data from that stream. There are two kinds of streams in PSDL, *dataflow* and *sampled streams*. *Dataflow* streams act as FIFO buffers, where the data values cannot be lost or replicated. These streams are used to synchronize data from multiple sources. Consumers of dataflow streams never read an empty stream. Similarly, each value in a stream is read only once. The control constraint used by the PSDL to distinguish a stream as dataflow is "TRIGGERED BY ALL".

Sampled Streams act as atomic memory cells providing continuous data. Connected operators can write on or read from the streams at uncoordinated rates. Older data are lost if the producer is faster than the consumer. Absence of "TRIGGERED BY ALL" control constraint implies the stream is sampled.

If any of the streams have any initial value, then it is known as *State Stream*. State Streams are declared in specification of the parent operator and are represented by thicker lines in the PSDL graph. State streams correspond to spaces that contain objects intended to be updated.

The mapping of dataflow streams or sampled streams into space-based communication is accomplished by treating the services, which in this case are the communication streams as objects to be shared.

4.2 Benefit of Loosely Coupled Communication

In tightly coupled systems, the communication process needs the answers to the questions of "who" to send to, "where" the receiving parties are located, and "when" the messages need to be sent. The "who" is which processes, "where" is which machines, and "when" is right now or later. They must be specified explicitly in order for the message to be delivered. Hence, in a distributed environment, in order for a producer and consumer to communicate successfully, they must know each other's identity and location, and must be running at the same time. This tight coupling leads to inflexible applications that are not mobile and in particular difficult to build, debug and change. In loosely coupled systems the issues of "who?", "where?" and "when?" are answered with "anyone", "anywhere" and "anytime".

"Anyone": Producers and consumers do not need to know each other's identities, but can instead communicate anonymously. In the sampled stream mapping, the producers place a message entity into the space without knowing who will be reading the messages. Similarly, the consumers read the message entity from the space without concern with the identity of the producers.

"Anywhere": Producers and consumers can be located anywhere, as long as they have access to an agreed-upon space for exchanging messages. The producer does not need to know the consumer's location. Conversely, the consumer picks up the message from the space using associative lookup, and has no need to be aware of the producer location. This is especially useful when the producers and the receivers roam from machine to machine, because the space-based programs do not need to change.

"Anytime": With space-based communication, producers and consumers are able to communicate even if they do not exist at the same time, because message entries persist in the space. This works well when the producers and the consumers operate asynchronously (Sampled Stream). This does not mean that synchronous communication would not work; the space is also an event driven repository and can trigger the consumers whenever new entities are created in the space. This decoupling in time is useful because it enables operators to be scheduled flexibly to accommodate real-time constraints.

5 How AICG Unifies Localized and Distributed Systems

The AICG model aims at bridging the differences between localized and distributed systems by simplifying the distributed model and encapsulating all the necessary elements of the distributed systems into the wrapper interfaces.

5.1 Localized and Distributed Systems

The major differences between localized and distributed systems concern the areas of latency, memory access, partial failure, and concurrency. Most of interoperability techniques try to hide the network and simplify the problems by stating that locations

of the software components do not affect the correctness of the computations, just the performance. These techniques concentrate on addressing the packing of data into portable forms, causing an invocation of a remote method somewhere on the network and so forth. However, latency, performance, partial failure and concurrency are some of the characteristics of distributed systems which also need serious attention.

5.1.1 Latency and Memory Access

The most obvious difference between accessing a local object and accessing a remote object has to do with the latency of the two calls. The difference between the two is currently between four and five orders of magnitude. In the AICG model vision of unified object where remote access is actually a three steps process, step one retrieves remote object from the space, step two executes the method of the remote object locally and lastly step three returns the object back to the space if it is modified. Developers must be aware of the latency and performance concerns. To ensure that the developers are aware of the issues, the AICG model requires the developers to specify the maximum latency period before an exception is raised. This forces the developers to consider the latency issues for the type of data and methods that are to be shared.

Another fundamental difference between local and remote computing concerns access to memory, specifically in the use of pointers. Simply stated, pointers are valid only within the local address space. There are two solutions; either all the memory access must be controlled by the underlying system, or the developers must be aware of the different type of access, whether local or remote.

Using the object-oriented paradigm to the fullest is a way of eliminating the boundary between the local and remote computing. However, it requires the developers to build an application that is entirely object-oriented. Such a unified model is difficult to enforce. The AICG solution to this issue is by enforcing the object-oriented paradigm only on distributed objects. The distributed object wrapper generated automatically forces all access to the actual shared object to go through the wrapper which is always a local object, eliminating direct reference to the actual object itself. This promotes and enforces the principle that "remote access and local access are exactly the same".

5.1.2 Partial Failure and Concurrency

In case of local systems, failures are usually total, affecting all the components of the system working together in an application. In distributed systems; one subsystem can fail while other systems continue. Similarly, a failure of network link is indistinguishable from the failure of a system on the other end of the link. The system may still function with partial failure, if certain unimportant components have crashed. It is however difficult to detect partial failure since there is no common agent that is able to determine which systems have failed, and this may result in the entire system going into unstable states

The AICG model uses the loosely-coupled paradigm, and component failure may have impact on the distributed system when the systems retrieve objects from the space and later crash before returning the objects back to space. The AICG model resolves this issue by enforcing update of distributed objects with transaction control

and allowing the developers to specify useful lifetime or lease for the object. When a lease has expired, the object would be automatically removed from the space.

Distributed objects by their nature must handle concurrent access and invocations. Invocations are usually asynchronous and difficult to model in distributed systems. Usually most models leave the concurrency issues to the developers discretion during implementation. However, this should be an interface issue and not solely an implementation issue, since dealing with concurrency can take place only by passing information from one object to another through the agency of the interface. The AICG model handles concurrency by design since there is only one copy of distributed object at a time in the entire distributed system. Processes are made to wait if the shared objects are not available in the space.

5.2 Transaction

Transaction control must validate operations to ensure consistency of the data, particularly when there are consistency constraints that link the states of several objects. The AICG model implements the transaction feature with the Jini Transaction model and provide a simplified interface for the developers.

5.2.1 Jini Transaction Model

All transactions are overseen by a transaction manager. When a distributed application needs operations to occur in a transaction secure manner, the process asks the transaction manager to create a transaction. Once a transaction has been created, one or more processes can perform operations under the transaction. A transaction can complete in two ways. If a transaction commits successfully, then all operations performed under it are complete. However, if problems arise, then the transaction is aborted and none of the operations occur. These semantics are provided by a two-phase commit protocol that is performed by the transaction manager as it interacts with the transaction participants.

5.2.2 AICG Transaction Model

AICG model encapsulates and manages the transaction procedures. All operations on a distributed object can be either with transaction control or without. Transaction control operations are controlled with a default lease of six sec. This default value of leasing time may, however, be overridden by the user. This is kept by the transaction manager as a leased resource, and if a lease expires before the operation committed, the transaction manager aborts the transaction.

The AICG model by default, enables all transactions for *write* operations with a transaction lease time of six seconds. The developer can modify the lease time through the PSDL SPACE *transactiontime* property.

All the read operations in the AICG model do not have transactions enabled by default. However, the user can enable it by using the property transactiontime with the upper limit in transaction time for the read operation.

5.3 Object Life Time (Leases/Timeout)

Leasing provides a methodology for controlling the life span of the distributed objects in the AICG space. This allows resources to be freed after a fixed period. This model is beneficial in the distributed environment, where partial failure can cause holders of resources to fail thereby disconnecting them from the resources before they can explicitly free them. In the absence of a leasing model, resource usage could grow without bound.

There are other constructive ways to harness the benefit of the leasing model besides using it as a garbage collector. For example, in a real-time system, the value of the information regarding some distributed objects becomes useless after certain deadlines. Accessing obsolete information can be more damaging in this case. By setting the lease on the distributed object, the AICG model automatically removes the object once the lease expires or the deadline is reached.

Java Spaces allocate resources that are tied to leases. When a distributed object is written into a space, it is granted a lease that specifies a period for which the space guarantees its storage. The holder of the lease may renew or cancel the lease before it expires. If the leaseholder does neither, the lease simply expires, and the space removes the entry from its store.

Generally, a distributed object that is not a part of a transaction lasts forever as long as the space exists, even if the leaseholder (the process that creates the object) has died. This configuration is enabled by setting the *SPACE lease* property in the Implementation to 0.
In real-time environment, a distributed object lasts for a fixed duration of x ms specified by the object designer. To keep the object alive, a write operation must be performed on the object before the lease expires. This configuration is set through the *SPACE lease* property in the Implementation to the time in ms required.

If an object has a lifetime, it must be renewed before it expires. In the AICG model, renewal is achieved by calling any method that modifies the object. If no modification is required, the developer can provide a dummy method with the spacemode set to "write". Invoking that method will automatically renew the lease.

5.4 AICG Event Notification

In a loosely-coupled distributed environment, it is desirable for an application to react to changes or arrival of newly distributed objects instead of "busy waiting" for it through polling. AICG provides this feature by introducing a callback mechanism that invokes user-defined methods when certain conditions are met.

Java provides a simple but powerful event model based on event sources, event listeners and event objects. An event source is any object that "fires" an event, usually based on some internal state change in the object. In this case, writing an object into space would generate an event. An event listener is an object that listens for events fired by an event source. Typically, an event source provides a method whereby

listeners can request to be added to a list of listeners. Whenever an event source fires an event, it notifies each of its registered listeners by calling a method on the listener object and passing it an event object.

Within a Java Virtual machine (JVM), an application is guaranteed that it will not miss an event fired from within. Distributed events on the other hand, had to travel either from one JVM to another JVM within a machine or between machines networked together. Events traveling from one JVM to another may be lost in transit, or may never reach their event listener. Likewise, an event may reach its listener more than once.

Space-based distributed events are built on top of the Jini Distributed Event model, and the AICG event model further extends it. When using the AICG event model, the space is an event source that fires events when entries are written into the space matching a certain template an application is interested in. When the event fires, the space sends a remote event object to the listener. The event listener codes are found in one of the generated AICG interface wrapper files. Upon receiving an event, the listener would spawn a new thread to process the event and invoke the application callback method. This allows the application codes to be executed without involving the developer in the process of event-management.

The distributed objects must have the SPACE properties for *Notification* set to yes. One of the application classes must *implement* (java term for inherit) the notifyAICG abstract class. The notifyAICG class has only one method, which is the callback method. The user class must override this method with the codes that need to be executed when an event fires.

6 Developing Distributed Application with the AICG Tool

This section describes the steps for developing distributed applications using the AICG model. An example of a C4ISR application is introduced in section 6.2 to aid the explanation of the process.

6.1 Development Process

The developer starts the development process by defining shared objects using the Prototyping System Description Language (PSDL). The PSDL is processed through a code generator (PSDLtoSpace) to produce a set of interface wrapper codes. The interface wrappers contain the necessary codes for interaction between application and the space without the need for the developers to be concerned with the writing and removing of objects in the space. The developers can treat shared or distributed objects as local objects, where synchronization and distribution are automatically handled by the interface codes.

6.2 Input Definition to the Code Generator

The following example demonstrates the development of one of the many distributed objects in a C4ISR system. Airplane positions picked up from sensors are processed to produce track objects. These objects are distributed over a large network and used by several clients' stations for displaying the positions of planes. Each track or plane is identified by track number. The tracks are 'owned' by a group of track servers, and only the track servers can update the track positions and its attributes. The clients only have read access on the track data. PSDL codes define (1) track object and as well as (2) Track_list object with the corresponding methods. AICG has used an extended version of the original PSDL grammar to model the interactions between applications in an entire distributed system.

The track PSDL starts with the definition of a *type* called **track**. It has only one identification field **tracknumber**. Of course, the **track** objects can have more than one field, but only one field is used in this case to uniquely identify any particular **track** object. The type **track_list** on the other hand, does not need an identification field since there is only one **track_list** object in the whole system. **Track_list** is used to keep a list of the **tracknumbers** of all the active tracks in the system at each moment in time.
All the operators (methods) of the *type* are defined immediately after the specification. Each method has a list of *input* and *output* parameters that define the arguments of the method. The most important portion in the method declaration is the *implementation*. The developer must be able to define the type of operation the method supposed to perform. The operation types are *constructor* (used to initialize the class), *read* (no modification to any field in the class) and *write* (modification is done to one or more fields in the class). These are necessary, as the code generated will encapsulate the synchronization of the distributed objects.

The other field in the implementation portion of the method, is *transactiontime*. *transactiontime* defines the upper limit in milliseconds within which the operation must be completed.

Upon running the example through the generator tool, a set of Java interface wrapper files are produced. Developers can ignore most of the generated files except the following:

- Track.java: this file contains the skeleton of the fields and the methods of the track class. The user is supposed to fill the body of the methods.
- TrackExtClient.java: this is the wrapper class that the client initializes and uses instead of the track class.
- TrackExtServer.java: this is the wrapper class that the server initializes and uses instead of the track class.
- NotifyAICG.java : this class must be extended or implemented by the application if event-notification and call-back are needed.

The methods found in the trackExtClient and trackExtServer have the same method names and signatures as the track class. In fact, the track class methods are called within trackExtClient or trackExtServer.

7 AICG Wrapper Design

This section explains the design of the AICG and the codes that are generated from psdl2java program.

7.1 AICG Wrapper Architecture

The AICG wrapper codes generated consists of four main module types. They are the Interface modules, the Event modules, Transaction modules and the Exception module. The interface modules implement the distributed object methods and communicate directly with the application. In reference to the example in section 6.2, the interface modules are entryAICG, track, trackExt, trackExtClient, trackExtServer. Instead of creating the actual object (track), the application should instantiate the corresponding interface object, either the trackExtClient or trackExtServer. Event modules (eventAICGID, eventAICGHandler, notifyAICG) handle external events generated from the JavaSpace that are of interest to the application. Transaction modules (transactionAICG, transactionManagerAICG) support the interface module with transaction services. Lastly, the exception module (exceptionAICG) defines the possible types of exceptions that can be raised and need to be captured by the application.

Each time the application instantiates a track class by creating a new trackExtServer, the following events take place in the Interface:

1. An Entry object is created together with the track object by the trackExtServer. The tack object is placed into the Entry object and stored in the space.
2. Transaction Manager is enabled.
3. The reference pointer to trackExtServer is returned to the application.

Each time a method (getID, getCallsign, getPosition) that does not modify the contents of the object is invoked, the following events take place in the Interface:

1. The application invokes the method through the Interface (trackExtServer/trackExtClient).
2. The Interface performs a Space "get" operation to update the local copy.
3. The method is then executed on the updated copy of the object to return the value back to the application.

Each time a method (setCallsign, setPosition), which does modify the contents of the object is invoked, the following events take place in the Interface:

1. The application invokes the method through the Interface.
2. The interface performs a Space "take" operation, which retrieves the object from the space.
3. The actual object method is then invoked to perform the modification.

4. Upon completion of the modification, the object is returned to the space by the interface using a "write" operation.

7.2 Interface Modules

The interface modules consist of the following modules; an entry (entryAICG) that is stored in space, the actual object (trackExt) that is shared and the object wrapper (trackExt, trackExtClient, trackExtServe.).

7.2.1 Entry

A space stores *entries*. An entry is a collection of typed objects that implements the Entry interface. The Entry interface is empty; it has no methods that have to be implemented. Empty interfaces are often referred to as "marker" interfaces because they are used to mark a class as suitable for some role. That is exactly what the Entry interface is used for, to mark a class appropriate for use within a space.

All entries in the AICG extend from this base class. It has one main public attribute, an identifier and an abstract method that returns the object. Any type of object can be stored in the entry. The only limitation is that the object must be serializable. The serializable property allows the java virtual machine to pass the entire object by value instead of by reference

All Entry attributes are declared as publicly accessible. Although it is not typical of fields to be defined as public in object-oriented programming style, an associative lookup is the way the space-based programs locate entries in the space. To locate an object in space, a template is specified that matches the contents of the fields. By declaring entry fields public, it allows the space to compare and locate the object. AICG encourages object-oriented programming style by encapsulating the actual data object into the entry. The object attributes can then be declared as private and made accessible only through clearly defined public methods of the object.

7.2.2 Serialization

Each distributed interface object is a local object that acts as a proxy to the remote space object. It is not a reference to a remote object but instead a connection passes all operations and value through the proxy to the remote space. All the objects must be serializable in order to meet this objective. The Serializable interface is "marker" interface that contains no methods and serves only to mark a class as appropriate for serialization. Classes marked as serializable should not contain pointers in their representation.

7.2.3 The Actual Object

We now look at the actual objects that are shared between servers and clients. The psdl2java generates a skeleton version of the actual class with the method names and its arguments. The bodies of the methods and its fields need to be filled by the developers.

7.2.4 Object Wrapper

Wrapping is an approach to protecting legacy software systems and commercial off-the-shelf (COTS) software products that require no modification of those products [1]. It consists of two parts, an adapter that provides some additional functionality for an application program at key external interfaces, and an encapsulation mechanism that binds the adapter to the application and protects the combined components [1].

In this context, the software being protected contains the actual distributed objects, and the AICG model has no way of knowing the behaviors of the distributed objects other than the operation types of of the methods. The adapter intercepts all invocations to provide additional functionalities such as synchronization between the local and distributed object, transaction control, event monitoring and exception handling. The encapsulation mechanism has been explained in the earlier section (AICG Architecture). Instead of instantiation of the actual object, the respective interface wrapper is instantiated. Instantiating the interface wrapper indirectly instantiates the actual object as well as storing the object in the space.

Three classes are generated for every distributed object. There are named with the object name appended with the following Ext, ExtClient, and ExtServer.

7.3 Event Modules

The event modules consist of the event callback template (notifyAICG), the event handler (eventAICGHandler) and the event identification object (eventAICGID).

7.3.1 Event Identification Object

The event identification object is used to distinguish one event from others. When an event of interest is registered, an event identification object is created to store the identification and event source. The object has only two methods, an 'equals' method that checks if two event identification objects are the same and a 'to string' method which is used by the event handler for retrieving the right event objects from the hash table.

7.3.2 Event Handler

Event Handler is the main body of the event operation in the AICG model. It handles registration of new events, deletion of old events, listening for event and invoking the right callback for that event. Inside the event handler are in fact, three inner classes to perform the above functions. Events are stored in a hash table with the event identification object as the key to the hash table. This allows fast retrieval of the event object and the callback methods.

The event handler listens for new events from the space or other sources. When an object is written to the space, an event is created by the space and captured by the all the listeners. The event handler would immediately spawn a new thread and check whether the event is of interest to the application.

7.3.3 The Callback Template

The callback template is a simple interface class with an abstract method listenerAICGEvents. Its main function is to allow the AICG model to invoke the application program when certain events of interest is "fired". As explained earlier, the notifyAICG interface needs to be implemented by each application that wishes to have notification.

7.4 The Transaction Modules

The transaction modules consist of a transaction interface (transactionAICG) and the transaction factory (transactionManagerAICG).

The transaction interface is a group of static methods that are used for obtaining references to the transaction manager server somewhere on the network. It uses the Java RMI registry or the look-up server to locate the transaction server.

The transaction factory uses the transaction interface to obtain the reference to the server, which is then used to create the default transaction or user-defined transactions. In short the transaction factory can perform the following:

1. Invoke the transaction interface to obtain a transaction manager.
2. Create a default transaction with lease time of 6 seconds.
3. Create a transaction with a user defined lease time.

7.5 The Exception Module

The exception module defines all the exception codes that are returned to the application when certain unexpected conditions occur in the AICG model. The exceptions include

- "UnDefinedExceptionCode"; unknown error occur.
- "SystemExceptionCode"; system level exceptions, such disk failure, network failure.
- "ObjectNotFoundException"; the space does not contain the object.
- "TransactionException"; transaction server not found, transaction expired before commit.
- "LeaseExpiredException"; object lease has expired.
- "CommunicationException"; space communication errors.
- "UnusableObjectException"; object corrupted.
- "ObjectExistsException"; there another object with the same key in the space.
- "NotificationException"; events notification errors.

8 Conclusion

The AICG vision of distributed object-oriented computing is an environment in which, from the developer's point of view, there is no distinct difference between

sharing of objects within an address space and objects that are on different machines. The model takes care of underlying interoperability issues by taking into account network latency, partial failure and concurrency. Automating the generation of interface wrappers directly from the Prototype System Description Language further enhances the reliability of the systems by enforcing proper object-oriented programming styles on the shared objects. Usage of PSDL for specification of shared objects also results in increased efficiency and shorter development time.

References

1. Valdis Berzins, Luqi, Bruce Shultes, Jiang Guo, Jim Allen, Ngom Cheng, Karen Gee, Tom Nguyen, and Eric Stierna : Interoperability Technology Assessment for Joint C4ISR Systems. Naval Postgraduate School Report NPSCS-00-001 September, (1999).
2. Nicholas Carriero, David Gelernter : How to Write Parallel Programs: A Guide to the Perplexed. ACM Computing Surveys, September (1989) 102-122.
3. David Gelernter : Generative Communication in Linda. ACM Transaction on Programming Languages and Systems, Vol. 7, No. 1, January (1985) 80-112.
4. Bill Joy : The Jini Specification. Addison Wesley, Inc. (1999)
5. Edward Keith : Core Jini. Prentice Hall, PTR, (1999)
6. Eun-Gyung Kim : A Study on Developing a Distributed Problem Solving System. IEEE Software, January (1995) 122-127
7. Fred Kuhns, Carlos O'Ryan, Douglas Schmidt, Ossama Othman, Jeff Parsons : The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware. IFIP 6th International Workshop on Protocols For High-Speed Networks (PfHSN' 99), August 25-27, (1999)
8. David Levein, Sergio Flores-Gaitan, Douglas Schmidt : An Empirical Evaulation of OD Endsystem Support for Real-time CORBA Object Request Brokers. Multimedia Computing and Network 2000, January (2000).
9. Luqi, Valdis Berzins : Rapidly Prototyping Real-Time Systems. IEEE Software, September (1988) 25-35
10. Luqi, Valdis Berzins, Bernd Kraemer, Laura White : A Proposed Design for a Rapid Prototyping Language. Naval Postgraduate School Technical Report, March (1989)
11. Luqi, Mantak Shing : CAPS – A Tool for Real-Time System Devleopment and Acquisition. Naval Research Review, Vol 1 (1992) 12-16
12. Luqi, Valdis Berzins, Raymond Yeh : A Prototyping Language for Real-Time Software. IEEE Software, October (1998) 1409-1423
13. Kevin Sullivan, Mark Marchukov, John Socha : Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model. IEEE Transactions on Software Engineering, Vol. 25, No. 4, July/August (1999) 584-599
14. Antoni Wolski : LINDA: A System for Loosely Integrated DataBases. IEEE Software, January (1989)66-73.